

Tugas Kecil 3 IF2211 Strategi Algoritma  
Semester II tahun 2023/2024

**Penyelesaian Permainan Word Ladder Menggunakan Algoritma UCS,  
Greedy Best First Search, dan A\***



Disusun oleh:

Mohammad Akmal Ramadan

13522161

**PROGRAM STUDI TEKNIK INFORMATIKA  
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA  
INSTITUT TEKNOLOGI BANDUNG  
2024**

## DAFTAR ISI

DAFTAR ISI.....	2
BAB 1	
ANALISIS DAN IMPLEMENTASI.....	3
1.1 Implementasi.....	3
1.2 Definisi $f(n)$ , $g(n)$ , dan $h(n)$ .....	3
1.3 Admissible pada Algoritma A*.....	4
1.4 Karakteristik Algoritma UCS dan BFS.....	4
1.5 Perbandingan Efisiensi Algoritma A* dengan UCS.....	4
1.6 Solusi Optimal pada Algoritma Greedy Best First Search.....	5
BAB 2	
SOURCE CODE PROGRAM.....	6
1. WordDictionaryParser.java.....	6
2. Main.java.....	8
3. Node.java.....	12
4. Ucs.java.....	13
5. Gbfs.java.....	16
6. Astar.java.....	18
BAB 3	
TEST CASE.....	22
BAB 4	
ANALISIS.....	31
BAB 5	
PRANALA REPOSITORY.....	34
LAMPIRAN.....	35

# BAB 1

## ANALISIS DAN IMPLEMENTASI

### 1.1 Implementasi

Implementasi pada UCS, GBFS, dan A\* tidak berbeda jauh, ketiganya menggunakan PriorityQueue untuk mensortir node yang akan dikunjungi terlebih dahulu sesuai dengan prioritas masing-masing algoritma. Program akan memiliki while loop yang akan mengiterasi semua isi priorityqueue atau sampai goal ditemukan. Lalu untuk node yang diiterasi akan dicari semua child nya (semua kata di dictionary yang memiliki maksimal 1 huruf berbeda dengan parent nya) lalu dihitung cost untuk masing-masing child, lalu ditambahkan ke priorityqueue, selanjutnya akan melanjutkan iterasi nodes dari priority queue yang memiliki heuristic value tertinggi.

Terdapat file `world_ladder.ser` yang menyimpan objek map berupa semua connection dari parent ke childnya, misal `make` memiliki anak `made`, `bake`, dan `cake`, sehingga key pada map adalah `make` dan values nya adalah `made`, `bake`, dan `cake`, begitu juga dengan kata lain yang ada pada dictionary ini. Extension `.ser` di sini berguna agar saat menjalankan program, program tidak perlu membuat map connection dari awal sehingga dapat langsung membaca dari file `.ser` yang sudah dibuat sebelumnya, yang mana melalui eksperimen yang sudah dilakukan, waktu yang dibutuhkan untuk membuat map untuk semua kata pada dictionary adalah sekitar 15 detik, dan ketika membaca map yang sudah jadi hanya membutuhkan sekitar 6 detik.

Penjelasan implementasi lebih lanjut akan dijelaskan di bab 2.

### 1.2 Definisi $f(n)$ , $g(n)$ , dan $h(n)$

$g(n)$  adalah ongkos yang dibutuhkan dari *root* ke *node*  $n$ ,  $g(n)$  dipakai dalam algoritma Uniform Cost Search (UCS).  $h(n)$  adalah ongkos yang dibutuhkan dari *node*  $n$  ke *goal*,  $h(n)$  dipakai dalam algoritma Greedy Best First Search.  $f(n)$  merupakan penjumlahan dari  $g(n)$  dan  $h(n)$ , yaitu  $f(n) = g(n) + h(n)$ ,  $f(n)$  dipakai oleh algoritma A\* yang memanfaatkan dua ongkos, yaitu  $g(n)$  dan  $h(n)$ .

### 1.3 Admissible pada Algoritma A\*

Heuristic yang digunakan pada algoritma A\* di permainan *word ladder* ini termasuk admissible, karena memenuhi aturan  $h(n) \leq h^*(n)$ , yaitu ongkos sebenarnya untuk mencapai *goal* tidak akan pernah kurang dari jarak Hamming atau jumlah karakter yang berbeda dari *node* ke *goal*, karena setiap perubahan kata hanya dapat merubah maksimal satu huruf sehingga jumlah ongkos untuk merubah n huruf ke goal membutuhkan minimal n ongkos atau langkah, contohnya untuk mengubah kata “COW” ke “CAT” dengan jarak Hamming sebesar 2, dibutuhkan minimal 2 langkah untuk mengubahnya, yaitu dari “COW” ke “COT” lalu ke “CAT”, tidak mungkin bisa kurang dari 2 langkah karena dari “COW” tidak bisa langsung ke “CAT” karena merubah 2 huruf dalam 1 langkah yang mana melanggar aturan dari permainan *word ladder*.

### 1.4 Karakteristik Algoritma UCS dan BFS

Sebenarnya UCS merupakan algoritma ekstensi dari BFS, yang membedakan kedua algoritma ini adalah UCS mengambil node yang memiliki ongkos terendah sedangkan BFS tidak memperhatikan ongkos dalam pengambilan node. Pada permainan *word ladder*, ongkos dari *root* akan selalu sama untuk semua nodes pada kedalaman tertentu, maka dari itu urutan node dan path yang dihasilkan oleh UCS akan sama dengan BFS pada kasus *word ladder*.

### 1.5 Perbandingan Efisiensi Algoritma A\* dengan UCS

A\* dapat dengan cepat menemukan solusi dengan biaya terendah dalam kasus *word ladder*. Ini karena heuristik jarak Hamming memberikan perkiraan yang baik tentang biaya untuk mencapai tujuan, yang memungkinkan A\* untuk mengarahkan pencarian ke arah tujuan dan menghindari area dari ruang pencarian yang tidak mungkin menghasilkan solusi optimal. Sementara itu, UCS tidak menggunakan heuristik dan hanya mempertimbangkan biaya untuk mencapai node saat ini (gCost). Ini berarti UCS harus mengeksplorasi lebih banyak node dibandingkan A\* sebelum menemukan solusi, yang membuatnya kurang efisien dalam kasus *word ladder*.

## 1.6 Solusi Optimal pada Algoritma Greedy Best First Search

Secara teoritis, algoritma Greedy Best-First Search (GBFS) tidak selalu menjamin solusi yang optimal, termasuk untuk persoalan word ladder.

Algoritma GBFS memilih node berikutnya untuk dikunjungi berdasarkan nilai heuristiknya saja, yang merupakan perkiraan biaya dari node saat ini ke tujuan. GBFS tidak mempertimbangkan biaya yang telah dikeluarkan untuk mencapai node saat ini. Oleh karena itu, GBFS bisa saja memilih jalur yang tampaknya mengarah langsung ke tujuan, tetapi pada akhirnya memiliki biaya total yang lebih tinggi.

Dalam konteks word ladder, ini berarti GBFS mungkin memilih transformasi kata yang tampaknya mendekatkan kita ke kata tujuan (berdasarkan heuristik), tetapi pada akhirnya memerlukan lebih banyak langkah transformasi.

Sebagai contoh, katakanlah kita ingin berubah dari kata "lead" ke "gold", dan kita bisa mengubah satu huruf pada suatu waktu. GBFS mungkin memilih untuk mengubah "lead" menjadi "load" karena "load" lebih mirip dengan "gold" menurut heuristik (hanya berbeda satu huruf), tetapi ini mungkin bukan pilihan terbaik karena mungkin memerlukan lebih banyak langkah untuk berubah dari "load" ke "gold" dibandingkan dengan jalur lain.

Jadi, meskipun GBFS bisa menemukan solusi untuk persoalan word ladder, solusi tersebut tidak selalu optimal. Untuk menjamin solusi optimal, kita biasanya menggunakan algoritma seperti A\* yang mempertimbangkan baik biaya yang telah dikeluarkan untuk mencapai node saat ini (gCost) dan perkiraan biaya untuk mencapai tujuan (hCost).

## BAB 2

### SOURCE CODE PROGRAM

#### 1. WordDictionaryParser.java

```
package dict;

import java.io.*;
import java.util.*;

public class WordDictionaryParser {
    public static Set<String> parseDictionary(String filePath) {
        Set<String> wordDictionary = new HashSet<>();

        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String word;
            while ((word = reader.readLine()) != null) {
                wordDictionary.add(word.trim());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        return wordDictionary;
    }

    public static Map<String, Set<String>> buildWordLadder(Set<String> wordDictionary) {
        Map<String, Set<String>> wordLadder = new HashMap<>();

        for (String key : wordDictionary) {
            Set<String> connections = new HashSet<>();
            for (int i = 0; i < key.length(); i++) {
                for (char c = 'A'; c <= 'Z'; c++) {
                    StringBuilder newWord = new StringBuilder(key);

```

```

        newWord.setCharAt(i, c);
        if (wordDictionary.contains(newWord.toString()) &&
!newWord.toString().equals(key)) {
            connections.add(newWord.toString());
        }
    }
}
wordLadder.put(key, connections);
}

return wordLadder;
}
}

```

1. `parseDictionary(String filePath)`: Metode ini digunakan untuk membaca file teks yang berisi daftar kata dan mengembalikan Set dari kata-kata tersebut.

- Pertama, metode ini membuat Set kosong bernama `wordDictionary` untuk menyimpan kata-kata.
- Kemudian, metode ini membuka file teks yang diberikan oleh `filePath` menggunakan `BufferedReader` dan `FileReader`.
- Metode ini kemudian membaca file baris per baris. Untuk setiap baris, ia menghapus spasi di awal dan akhir baris dengan `trim()` dan menambahkan kata tersebut ke `wordDictionary`.
- Jika terjadi kesalahan saat membaca file, metode ini akan menangkap `IOException` dan mencetak stack trace.
- Akhirnya, metode ini mengembalikan `wordDictionary`.

2. `buildWordLadder(Set<String> wordDictionary)`: Metode ini digunakan untuk membangun struktur data yang disebut "word ladder" dari Set kata-kata yang diberikan. Word ladder adalah Map di mana setiap kata dipetakan ke Set kata-kata lain yang bisa dicapai dari kata tersebut dengan mengubah satu huruf.

- Pertama, metode ini membuat Map kosong bernama wordLadder untuk menyimpan word ladder.
- Kemudian, metode ini mengulangi setiap kata (key) dalam wordDictionary.
- Untuk setiap kata, metode ini membuat Set kosong bernama connections untuk menyimpan kata-kata yang bisa dicapai dari kata tersebut.
- Metode ini kemudian mengulangi setiap huruf dalam kata dan mencoba menggantinya dengan setiap huruf dari 'A' hingga 'Z' menggunakan StringBuilder.
- Jika kata baru yang dihasilkan ada dalam wordDictionary dan berbeda dari kata asli, metode ini menambahkannya ke connections.
- Setelah semua huruf dalam kata telah dicoba, metode ini menambahkan connections ke wordLadder dengan kata sebagai kunci.
- Akhirnya, metode ini mengembalikan wordLadder.

Secara keseluruhan, kelas WordDictionaryParser ini digunakan untuk membaca daftar kata dari file teks dan membangun struktur data yang memetakan setiap kata ke set kata-kata lain yang bisa dicapai dari kata tersebut dengan mengubah satu huruf. Struktur data ini bisa digunakan untuk mencari jalur terpendek antara dua kata, seperti dalam permainan word ladder.

## 2. Main.java

```
import java.io.*;
import java.util.*;

import algorithms.Astar;
import algorithms.Gbfs;
import algorithms.Ucs;
import dict.WordDictionaryParser;

public class Main {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        String dictionaryFilePath = "../dict/dictionary.txt";

        Set<String> wordDictionary =
WordDictionaryParser.parseDictionary(dictionaryFilePath);
        File graphFile = new File("../data/word_ladder.ser");
```



```

Map<String, Set<String>> wordLadder;

if (graphFile.exists()) {
    try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(graphFile))) {
        wordLadder = (Map<String, Set<String>>) ois.readObject();
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
        return;
    }
} else {
    wordLadder = WordDictionaryParser.buildWordLadder(wordDictionary);

    try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(graphFile))) {
        oos.writeObject(wordLadder);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

Scanner scanner = new Scanner(System.in);
int choice;

while (true) {
    System.out.println("1. UCS");
    System.out.println("2. Greedy BFS");
    System.out.println("3. A*");
    System.out.println("4. Exit");
    System.out.print("Choose the Algorithm: ");
    try {
        choice = scanner.nextInt();

        if (choice < 1 || choice > 4) {
            System.out.println("\nInvalid choice\n");
            continue;
        }

        if (choice == 4) {
            System.out.println("\nThank you for using the program.");

```

```

        System.out.println("#Semester4CepatKelarDong");
        System.out.println("#Udah6TubesNih");
        System.out.println("#TubesOSMasihKeyboard : (");
        scanner.close();
        System.exit(0);
    }

    System.out.print("\nEnter the initial word: ");
    String initial = scanner.next().toUpperCase();
    if (!initial.matches("[A-Za-z]+") || !wordDictionary.contains(initial)) {
        System.out.println("\nInvalid input. Please enter a word in the
dictionary.\n");
        continue;
    }
    System.out.print("Enter the goal word: ");
    String goal = scanner.next().toUpperCase();
    if (!goal.matches("[A-Za-z]+") || !wordDictionary.contains(goal))
{
        System.out.println("\nInvalid input. Please enter a word in the
dictionary.\n");
        continue;
    }
    if (initial.length() != goal.length()) {
        System.out.println("\nInvalid input. The initial and goal words
must have the same length.\n");
        continue;
    }
    List<String> result;
    switch (choice) {
        case 1:
            Ucs ucs = new Ucs();
            result = ucs.algorithms(initial, goal, wordLadder);
            printResult(result);
            break;
        case 2:
            Gbfs gbfs = new Gbfs();
            result = gbfs.algorithms(initial, goal, wordLadder);
            printResult(result);
            break;
    }
}

```

```

        case 3:
            Astar astar = new Astar();
            result = astar.algorithms(initial, goal, wordLadder);
            printResult(result);
            break;
    }
} catch (InputMismatchException e) {
    System.out.println("\nInvalid choice\n");
    scanner.nextLine();
    continue;
} catch (NoSuchElementException e) {
    System.out.println("\nInvalid choice\n");
    scanner.nextLine();
    continue;
}
}
}

public static void printResult(List<String> result) {
    if (result.size() <= 1){
        System.out.println("\nNo path found!\n");
    }
    else{
        System.out.println("Path: " + result + "\n");
        System.out.println("Number of steps: " + (result.size() -1) + "\n");
    }
}
}
}

```

1. Impor Kelas: Kode ini mengimpor kelas yang diperlukan dari paket algorithms dan dict.

2. Metode Main: Metode main adalah titik masuk utama program. Ini adalah metode pertama yang dijalankan ketika program dimulai.

- Pertama, metode ini mendefinisikan dictionaryFilePath yang merupakan lokasi file kamus kata.
- Kemudian, metode ini memanggil parseDictionary untuk membaca kamus kata dari file dan menyimpannya dalam Set bernama wordDictionary.

- Metode ini kemudian mencoba membaca wordLadder dari file serialisasi. Jika file tidak ada, metode ini memanggil buildWordLadder untuk membuat wordLadder dari wordDictionary dan menyimpannya ke file untuk penggunaan di masa mendatang.
- Setelah itu, metode ini memulai loop utama program, yang meminta pengguna untuk memilih algoritma pencarian dan memasukkan kata awal dan tujuan. Loop ini berlanjut sampai pengguna memilih untuk keluar.

3. Metode printResult: Metode ini digunakan untuk mencetak hasil pencarian jalur. Jika ukuran hasil kurang dari atau sama dengan 1, metode ini mencetak pesan bahwa tidak ada jalur yang ditemukan. Jika tidak, metode ini mencetak jalur dan jumlah langkah.

### 3. Node.java

```
package algorithms;

public class Node {
    String word;
    int gCost;
    int hCost;
    Node parent;

    Node(String word) {
        this.word = word;
        this.gCost = Integer.MAX_VALUE;
        this.hCost = 0;
        this.parent = null;
    }

    Node(String word, int gCost, int hCost) {
        this.word = word;
        this.gCost = gCost;
        this.hCost = hCost;
        this.parent = null;
    }

    int getTotalCost() {
        return gCost + hCost;
    }
}
```

```

    }

    int getHCost() {
        return hCost;
    }

    int getGCost() {
        return gCost;
    }
}

```

Ini merupakan class Node yang akan dipakai dalam algoritma UCS, GBFS, dan A\*. Ada ctor, cctor, dan juga getter untuk total, g, dan h cost.

#### 4. Ucs.java

```

package algorithms;

import java.util.*;

public class Ucs {
    public List<String> algorithms(String initial, String goal,
Map<String, Set<String>> wordLadder) {
        long startTime = System.currentTimeMillis(); // Start time
        Runtime runtime = Runtime.getRuntime();
        long usedMemoryBefore = runtime.totalMemory() -
runtime.freeMemory(); // Memory used before execution

        Set<String> visited = new HashSet<>();
        PriorityQueue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(Node::getGCost));
        Map<String, Node> nodes = new HashMap<>();
        Node startNode = new Node(initial, 0, 0);
        nodes.put(initial, startNode);
        queue.add(startNode);

        while (!queue.isEmpty()) {
            Node current = queue.poll();

```

```

        visited.add(current.word);
        if (current.word.equals(goal)) {
            break;
        }
        Set<String> connections = wordLadder.get(current.word);
        if (connections != null) {
            for (String connection : connections) {
                if (!visited.contains(connection)) {
                    Node nextNode = nodes.getOrDefault(connection, new
Node(connection));

                    nodes.put(connection, nextNode);
                    int newCost = current.gCost + 1;
                    if (newCost < nextNode.gCost) {
                        nextNode.gCost = newCost;
                        nextNode.parent = current;
                        queue.add(nextNode);
                    }
                }
            }
        }
    }

    LinkedList<String> path = new LinkedList<>();
    for (Node node = nodes.get(goal); node != null; node =
node.parent) {
        path.addFirst(node.word);
    }

    printPerformanceMetrics(startTime, runtime, usedMemoryBefore);

    return path;
}

private void printPerformanceMetrics(long startTime, Runtime runtime,
long usedMemoryBefore) {
    long stopTime = System.currentTimeMillis();
    long elapsedTime = stopTime - startTime;
    long usedMemoryAfter = runtime.totalMemory() -
runtime.freeMemory();

```

```

        System.out.println("\nExecution time: " + elapsedTime + "
milliseconds");
        System.out.println("Memory used: " + (usedMemoryAfter -
usedMemoryBefore) + " bytes");
    }
}

```

1. Metode algorithms: Metode ini menerima kata awal, kata tujuan, dan word ladder sebagai argumen, dan mengembalikan jalur terpendek dari kata awal ke kata tujuan.

- Pertama, metode ini mengambil waktu mulai dan memori yang digunakan sebelum eksekusi untuk mengukur kinerja.
- Metode ini kemudian membuat Set bernama visited untuk menyimpan kata-kata yang telah dikunjungi, PriorityQueue bernama queue untuk menyimpan node yang akan dikunjungi (dengan node yang memiliki gCost terendah diambil pertama), dan Map bernama nodes untuk menyimpan semua node yang telah dibuat.
- Metode ini kemudian membuat Node untuk kata awal dengan gCost 0 dan menambahkannya ke queue dan nodes.
- Metode ini kemudian memulai loop utama, yang berlanjut sampai queue kosong. Dalam setiap iterasi, metode ini mengambil node dengan gCost terendah dari queue, menambahkannya ke visited, dan jika node tersebut adalah tujuan, loop dihentikan.
- Jika node bukan tujuan, metode ini mendapatkan semua kata yang bisa dicapai dari kata saat ini dari wordLadder, dan untuk setiap kata tersebut, jika belum dikunjungi, metode ini membuat Node baru atau mendapatkan Node yang ada dari nodes, menghitung gCost baru, dan jika gCost baru lebih rendah dari gCost sebelumnya, metode ini memperbarui gCost dan parent dari Node dan menambahkannya ke queue.
- Setelah loop selesai, metode ini membuat LinkedList bernama path dan mengisi path dengan kata-kata dari node tujuan ke node awal dengan mengikuti parent dari setiap node.
- Metode ini kemudian mencetak metrik kinerja dan mengembalikan path.

2. Metode `printPerformanceMetrics`: Metode ini menerima waktu mulai dan memori yang digunakan sebelum eksekusi sebagai argumen, menghitung waktu berhenti dan memori yang digunakan setelah eksekusi, dan mencetak waktu eksekusi dan memori yang digunakan.

## 5. Gbfs.java

```
package algorithms;

import java.util.*;

public class Gbfs {
    public List<String> algorithms(String initial, String goal, Map<String,
Set<String>> wordLadder) {
        long startTime = System.currentTimeMillis();
        Runtime runtime = Runtime.getRuntime();
        long usedMemoryBefore = runtime.totalMemory() - runtime.freeMemory();

        Set<String> visited = new HashSet<>();
        PriorityQueue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(Node::getHCost));
        Map<String, Node> nodes = new HashMap<>();
        Node startNode = new Node(initial, 0, countHeuristic(initial, goal));
        nodes.put(initial, startNode);
        queue.add(startNode);

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            visited.add(current.word);
            if (current.word.equals(goal)) {
                break;
            }
            Set<String> connections = wordLadder.get(current.word);
            if (connections != null) {
                for (String connection : connections) {
                    if (!visited.contains(connection)) {
                        Node nextNode = nodes.getDefault(connection, new
Node(connection));
                        nextNode.parent = current; // Set the parent of the node
                        nextNode.hCost = countHeuristic(connection, goal);
```



```

        nodes.put(connection, nextNode);
        queue.add(nextNode);
    }
}

LinkedList<String> path = new LinkedList<>();
for (Node node = nodes.get(goal); node != null; node = node.parent) {
    path.addFirst(node.word);
}

printPerformanceMetrics(startTime, runtime, usedMemoryBefore);
return path;
}

private int countHeuristic(String current, String goal) {
    int count = 0;
    for (int i = 0; i < current.length(); i++) {
        if (current.charAt(i) != goal.charAt(i)) {
            count++;
        }
    }
    return count;
}

private void printPerformanceMetrics(long startTime, Runtime runtime,
long usedMemoryBefore) {
    long stopTime = System.currentTimeMillis();
    long elapsedTime = stopTime - startTime;
    long usedMemoryAfter = runtime.totalMemory() - runtime.freeMemory();
    System.out.println("\nExecution time: " + elapsedTime + "
milliseconds");
    System.out.println("Memory used: " + (usedMemoryAfter -
usedMemoryBefore) + " bytes");
}
}

```

1. Metode algorithms: Metode ini menerima kata awal, kata tujuan, dan word ladder sebagai argumen, dan mengembalikan jalur dari kata awal ke kata tujuan.

- Pertama, metode ini mengambil waktu mulai dan memori yang digunakan sebelum eksekusi untuk mengukur kinerja.
- Metode ini kemudian membuat Set bernama visited untuk menyimpan kata-kata yang telah dikunjungi, PriorityQueue bernama queue untuk menyimpan node yang akan dikunjungi (dengan node yang memiliki hCost terendah diambil pertama), dan Map bernama nodes untuk menyimpan semua node yang telah dibuat.
- Metode ini kemudian membuat Node untuk kata awal dengan hCost yang dihitung dengan countHeuristic dan menambahkannya ke queue dan nodes.
- Metode ini kemudian memulai loop utama, yang berlanjut sampai queue kosong. Dalam setiap iterasi, metode ini mengambil node dengan hCost terendah dari queue, menambahkannya ke visited, dan jika node tersebut adalah tujuan, loop dihentikan.
- Jika node bukan tujuan, metode ini mendapatkan semua kata yang bisa dicapai dari kata saat ini dari wordLadder, dan untuk setiap kata tersebut, jika belum dikunjungi, metode ini membuat Node baru atau mendapatkan Node yang ada dari nodes, menghitung hCost baru, dan jika hCost baru lebih rendah dari hCost sebelumnya, metode ini memperbarui hCost dan parent dari Node dan menambahkannya ke queue.
- Setelah loop selesai, metode ini membuat LinkedList bernama path dan mengisi path dengan kata-kata dari node tujuan ke node awal dengan mengikuti parent dari setiap node.
- Metode ini kemudian mencetak metrik kinerja dan mengembalikan path.

2. Metode countHeuristic: Metode ini menerima dua kata sebagai argumen dan mengembalikan jumlah posisi di mana dua kata tersebut memiliki karakter yang berbeda (jarak Hamming). Ini digunakan sebagai heuristik dalam algoritma GBFS.

## 6. Astar.java

```
package algorithms;  
  
import java.util.*;
```

```

public class Astar {
    public List<String> algorithms(String initial, String goal, Map<String,
Set<String>> wordLadder) {
        long startTime = System.currentTimeMillis();
        Runtime runtime = Runtime.getRuntime();
        long usedMemoryBefore = runtime.totalMemory() - runtime.freeMemory();

        Set<String> visited = new HashSet<>();
        PriorityQueue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(Node::getTotalCost));
        Map<String, Node> nodes = new HashMap<>();
        Node startNode = new Node(initial, 0, countHCost(initial, goal));
        nodes.put(initial, startNode);
        queue.add(startNode);

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            visited.add(current.word);
            if (current.word.equals(goal)) {
                break;
            }
            Set<String> connections = wordLadder.get(current.word);
            if (connections != null) {
                for (String connection : connections) {
                    if (!visited.contains(connection)) {
                        Node nextNode = nodes.getDefault(connection, new
Node(connection));
                        nodes.put(connection, nextNode);
                        int newCost = current.gCost + 1;
                        if (newCost < nextNode.gCost) {
                            nextNode.gCost = newCost;
                            nextNode.hCost = countHCost(connection, goal);
                            nextNode.parent = current;
                            queue.add(nextNode);
                        }
                    }
                }
            }
        }
    }
}

```

```

LinkedList<String> path = new LinkedList<>();
for (Node node = nodes.get(goal); node != null; node = node.parent) {
    path.addFirst(node.word);
}

printPerformanceMetrics(startTime, runtime, usedMemoryBefore);
return path;
}

private int countHCost(String current, String goal) {
    int count = 0;
    for (int i = 0; i < current.length(); i++) {
        if (current.charAt(i) != goal.charAt(i)) {
            count++;
        }
    }
    return count;
}

private void printPerformanceMetrics(long startTime, Runtime runtime,
long usedMemoryBefore) {
    long stopTime = System.currentTimeMillis();
    long elapsedTime = stopTime - startTime;
    long usedMemoryAfter = runtime.totalMemory() - runtime.freeMemory();
    System.out.println("\nExecution time: " + elapsedTime + "
milliseconds");
    System.out.println("Memory used: " + (usedMemoryAfter -
usedMemoryBefore) + " bytes");
}
}

```

1. Metode algorithms: Metode ini menerima kata awal, kata tujuan, dan word ladder sebagai argumen, dan mengembalikan jalur dari kata awal ke kata tujuan.

- Pertama, metode ini mengambil waktu mulai dan memori yang digunakan sebelum eksekusi untuk mengukur kinerja.
- Metode ini kemudian membuat Set bernama visited untuk menyimpan kata-kata yang telah dikunjungi, PriorityQueue bernama queue untuk menyimpan node yang akan

dikunjungi (dengan node yang memiliki totalCost terendah diambil pertama), dan Map bernama nodes untuk menyimpan semua node yang telah dibuat.

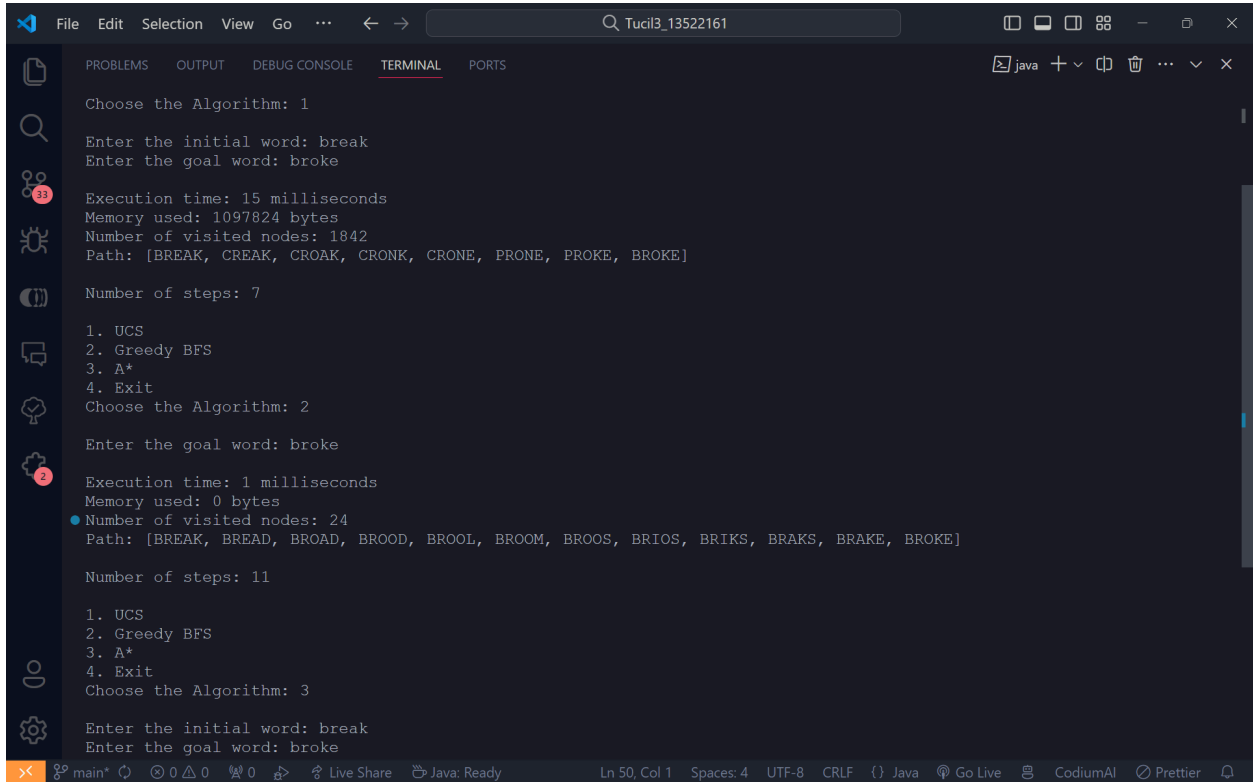
- Metode ini kemudian membuat Node untuk kata awal dengan gCost 0 dan hCost yang dihitung dengan countHCost dan menambahkannya ke queue dan nodes.
- Metode ini kemudian memulai loop utama, yang berlanjut sampai queue kosong. Dalam setiap iterasi, metode ini mengambil node dengan totalCost terendah dari queue, menambahkannya ke visited, dan jika node tersebut adalah tujuan, loop dihentikan.
- Jika node bukan tujuan, metode ini mendapatkan semua kata yang bisa dicapai dari kata saat ini dari wordLadder, dan untuk setiap kata tersebut, jika belum dikunjungi, metode ini membuat Node baru atau mendapatkan Node yang ada dari nodes, menghitung gCost dan hCost baru, dan jika totalCost baru lebih rendah dari totalCost sebelumnya, metode ini memperbarui gCost, hCost, dan parent dari Node dan menambahkannya ke queue.
- Setelah loop selesai, metode ini membuat LinkedList bernama path dan mengisi path dengan kata-kata dari node tujuan ke node awal dengan mengikuti parent dari setiap node.
- Metode ini kemudian mencetak metrik kinerja dan mengembalikan path.

2. Metode countHCost: Metode ini menerima dua kata sebagai argumen dan mengembalikan jumlah posisi di mana dua kata tersebut memiliki karakter yang berbeda (jarak Hamming). Ini digunakan sebagai heuristik dalam algoritma A\*.

## BAB 3

### TEST CASE

#### 1. BREAK -> BROKE



```
File Edit Selection View Go ... ← → 🔍 Tucil3_13522161
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Choose the Algorithm: 1
Enter the initial word: break
Enter the goal word: broke
Execution time: 15 milliseconds
Memory used: 1097824 bytes
Number of visited nodes: 1842
Path: [BREAK, CREAK, CROAK, CRONK, CRONE, PRONE, PROKE, BROKE]
Number of steps: 7
1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 2
Enter the goal word: broke
Execution time: 1 milliseconds
Memory used: 0 bytes
● Number of visited nodes: 24
Path: [BREAK, BREAD, BROAD, BROOD, BROOL, BROOM, BROOS, BRIOS, BRIKS, BRAKS, BRAKE, BROKE]
Number of steps: 11
1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 3
Enter the initial word: break
Enter the goal word: broke
```

```
File Edit Selection View Go ... ← → Tucil3_13522161
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Enter the goal word: broke

Execution time: 1 milliseconds
Memory used: 0 bytes
Number of visited nodes: 24
Path: [BREAK, BREAD, BROAD, BROOD, BROOL, BROOM, BROOS, BRIOS, BRIKS, BRAKS, BRAKE, BROKE]

Number of steps: 11
1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 3

Enter the initial word: break
Enter the goal word: broke

Execution time: 2 milliseconds
Memory used: 0 bytes
Number of nodes visited: 59
Path: [BREAK, CREAK, CROAK, CRONK, CRONE, CROME, BROME, BROKE]

Number of steps: 7
1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: █

main* 0 0 0 0 Live Share Java: Ready Ln 50, Col 1 Spaces: 4 UTF-8 CRLF {} Java Go Live CodiumAI Prettier
```

## 2. MAKE -> FLAT

```
File Edit Selection View Go ... ← → Tucil3_13522161
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Choose the Algorithm: 1

Enter the initial word: make
Enter the goal word: flat

Execution time: 12 milliseconds
Memory used: 2097136 bytes
Number of visited nodes: 4592
Path: [MAKE, MALE, MALT, MELT, FELT, FEAT, FLAT]

Number of steps: 6
1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 2

Enter the initial word: make
Enter the goal word: flat

Execution time: 0 milliseconds
Memory used: 0 bytes
Number of visited nodes: 11
Path: [MAKE, FAKE, FAZE, FACE, FACT, FAUT, FAST, FEST, FEAT, FLAT]

Number of steps: 9
1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 3

Enter the initial word: make

main* 0 0 0 0 Live Share Java: Ready Ln 50, Col 1 Spaces: 4 UTF-8 CRLF {} Java Go Live CodiumAI Prettier
```

```

File Edit Selection View Go ... ← → Tucil3_13522161
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
3. A*
4. Exit
Choose the Algorithm: 2

Enter the initial word: make
Enter the goal word: flat

Execution time: 0 milliseconds
Memory used: 0 bytes
Number of visited nodes: 11
Path: [MAKE, FAKE, FAZE, FACE, FACT, FAUT, FAST, FEST, FEAT, FLAT]

Number of steps: 9

1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 3

Enter the initial word: make
Enter the goal word: flat

Execution time: 1 milliseconds
Memory used: 0 bytes
Number of nodes visited: 216
Path: [MAKE, MASE, MAST, FAST, FEST, FEAT, FLAT]

Number of steps: 6

1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm:

```

### 3. CHARGE -> COMEDO

```

File Edit Selection View Go ... ← → Tucil3_13522161
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Choose the Algorithm: 1

Enter the initial word: charge
Enter the goal word: comedo

Execution time: 24 milliseconds
Memory used: 4325408 bytes
Number of visited nodes: 14537
Path: [CHARGE, CHANGE, CHANGS, CHINGS, CHINES, CHINED, COINED, CONNED, CONNES, CONIES, CONINS, CONING, HONING, HOMING, HOMINY, HOMILY, HOMELY, COMELY, COMEDY, COMEDO]

Number of steps: 19

1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 2

Enter the initial word: charge
Enter the goal word: comedo

Execution time: 3 milliseconds
Memory used: 880104 bytes
Number of visited nodes: 499
Path: [CHARGE, CHANGE, CHANGS, CHANKS, CHANTS, CHARTS, CHARDS, CHARES, CHORES, CHOKES, CHOKER, COOKER, COOLER, COOPER, COMPER, COMMER, COMBER, COMBED, COMBES, COMBOS, COMPOS, COMPTS, COMETS, COMERS, COERS, COPERS, CODERS, CORERS, COVERS, COVEYS, COVENS, COVINS, COVING, COMING, LOMING, DOMING, HOMING, HOMINY, HOMILY, HOMELY, COMELY, COMEDY, COMEDO]

Number of steps: 42

1. UCS
2. Greedy BFS
3. A*
4. Exit

```







```

TAXES, UNTAXED, UNWAXED, UNWAGED, UNWAKED, UNBAKED, UNBARED, UNBALED, UNBASED, UNCAGED, ENCASED, ENCAGED, ENGAGED, EN
GAGER, ENGAGEE, ENGAGES, ENRAGES, ENRAGED, ENRACED, UNRACED, UNPACED, UNPAGED, UNCAGED, INCAGED, INCAGES, INNAGES, EN
NAGES, ENCAGES, UNCAGES, UNCAPES, UNCAKES, UNCAGES, UNEASES, UREASES, CREASES, CREASED, CREATED, CLEATED, CHEATED, CH
EAPED, CHEAPEN, CHEAPER, CHEEPER, CHEERER, CHEERED, CHEESED, CHEEKED, CLEEKED, CLECKED, CHECKED, CHOCKED, CHOCKER, CL
OCKER, CLICKER, CLINKER, CLINGER, CLANGER, CHANGER, CHANNER, CHANTER, CHUNTER, COUNTER, COULTER, COURTER, COURIER, CO
URIED, COORIED, COORIES, COWRIES, CORRIES, CARRIES, CABRIES, CABBIES, CARBIES, CARNIES, CARVIES, CARDIES, CADDIES, CA
LKIES, TALLIES, WALLIES, BALLIES, BALLSES, BALASES, BALISES, WALISES, VALISES, VALINES, SALINES, SAVINES, RAVINES, RA
VINED, RAVENED, HAVENED, HAVERED, WAVERED, TAVERED, TABERED, TABORED, TABORET, TABARET, CABARET]

Number of steps: 126

1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 3

Enter the initial word: atlases
Enter the goal word: cabaret

Execution time: 19 milliseconds
Memory used: 1217048 bytes
Number of nodes visited: 6572
Path: [ATLASES, ANLASES, ANLACES, UNLACES, UNLACED, UNLADED, UNFADED, UNFAMED, UNTAMED, UNTIMED, UNLIMED, UNLIMES, UN
LINES, UNLINES, ONDINES, ONDINGS, ENDINGS, ENRINGS, EARINGS, EATINGS, RATINGS, RATINES, RAVINES, RAVINED, RAVENED, HA
VENED, HAVERED, TAVERED, TABERED, TABORED, TABORET, TABARET, CABARET]

Number of steps: 32

1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm:

```

## 6. BOYISH -> PAINCH

```

Choose the Algorithm: 1

Enter the initial word: boyish
Enter the goal word: painch

Execution time: 20 milliseconds
Memory used: 2097152 bytes
Number of visited nodes: 9856
Path: [BOYISH, TOYISH, TONISH, MONISH, MORISH, MARISH, PARISH, PARIAS, PARIAL, PARRAL, PARREL, PARRED, PAIRED, PAINED
, PAWNED, PAWNEE, PAWNCE, PAUNCE, PAUNCH, PAINCH]

Number of steps: 19

1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 2

Enter the initial word: boyish
Enter the goal word: painch

Execution time: 0 milliseconds
Memory used: 0 bytes
Number of visited nodes: 52
Path: [BOYISH, TOYISH, TONISH, MONISH, MORISH, MARISH, PARISH, PARIAS, PARIAN, PARIAL, PARRAL, PARREL, PARRED, PAIRED
, PAINED, PAWNED, PAWNER, PAWNEE, PAWNCE, PAUNCE, PAUNCH, PAINCH]

Number of steps: 21

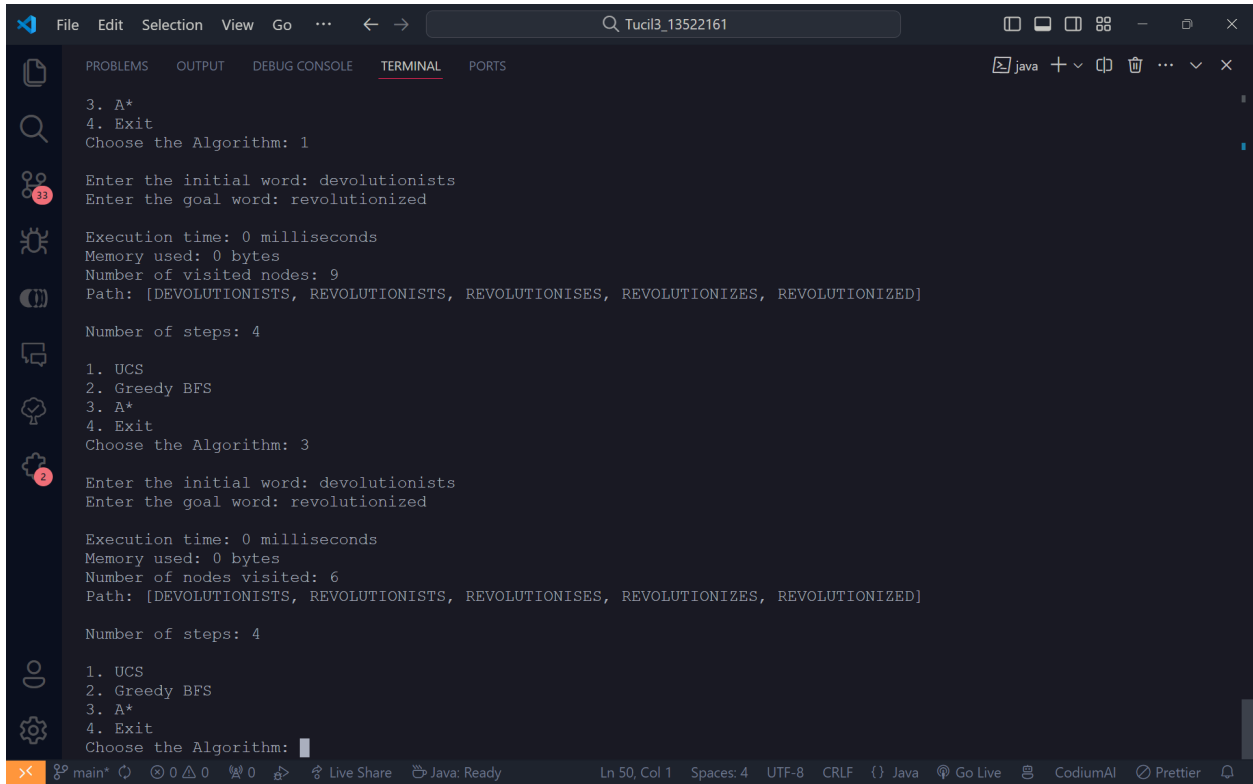
1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 3

```

```
File Edit Selection View Go ... ← → Tucil3_13522161
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Choose the Algorithm: 2
Enter the initial word: boyish
Enter the goal word: painch
Execution time: 0 milliseconds
Memory used: 0 bytes
Number of visited nodes: 52
Path: [BOYISH, TOYISH, TONISH, MONISH, MORISH, MARISH, PARISH, PARIAS, PARIAN, PARIAL, PARRAL, PARREL, PARRED, PAIRED, PAINED, PAWNED, PAWNER, PAWNEE, PAWNCE, PAUNCE, PAUNCH, PAINCH]
Number of steps: 21
1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 3
Enter the initial word: boyish
Enter the goal word: painch
Execution time: 0 milliseconds
Memory used: 880104 bytes
Number of nodes visited: 887
Path: [BOYISH, TOYISH, TONISH, MONISH, MORISH, MARISH, PARISH, PARIAS, PARIAN, PARRAL, PARREL, PARRED, PAIRED, PAINED, PAWNED, PAWNEE, PAWNCE, PAUNCE, PAUNCH, PAINCH]
Number of steps: 19
1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 
```

## 7. DEVOLUTIONISTS -> REVOLUTIONIZED

```
File Edit Selection View Go ... ← → Tucil3_13522161
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Choose the Algorithm: 1
Enter the initial word: devolutionists
Enter the goal word: revolutionized
Execution time: 0 milliseconds
Memory used: 0 bytes
Number of visited nodes: 9
Path: [DEVOLUTIONISTS, REVOLUTIONISTS, REVOLUTIONISES, REVOLUTIONIZES, REVOLUTIONIZED]
Number of steps: 4
1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 1
Enter the initial word: devolutionists
Enter the goal word: revolutionized
Execution time: 0 milliseconds
Memory used: 0 bytes
Number of visited nodes: 9
Path: [DEVOLUTIONISTS, REVOLUTIONISTS, REVOLUTIONISES, REVOLUTIONIZES, REVOLUTIONIZED]
Number of steps: 4
1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 3
Enter the initial word: devolutionists
```



```
File Edit Selection View Go ... ← → Q Tucil3_13522161
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
3. A*
4. Exit
Choose the Algorithm: 1

Enter the initial word: devolutionists
Enter the goal word: revolutionized

Execution time: 0 milliseconds
Memory used: 0 bytes
Number of visited nodes: 9
Path: [DEVOLUTIONISTS, REVOLUTIONISTS, REVOLUTIONISES, REVOLUTIONIZES, REVOLUTIONIZED]

Number of steps: 4

1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 3

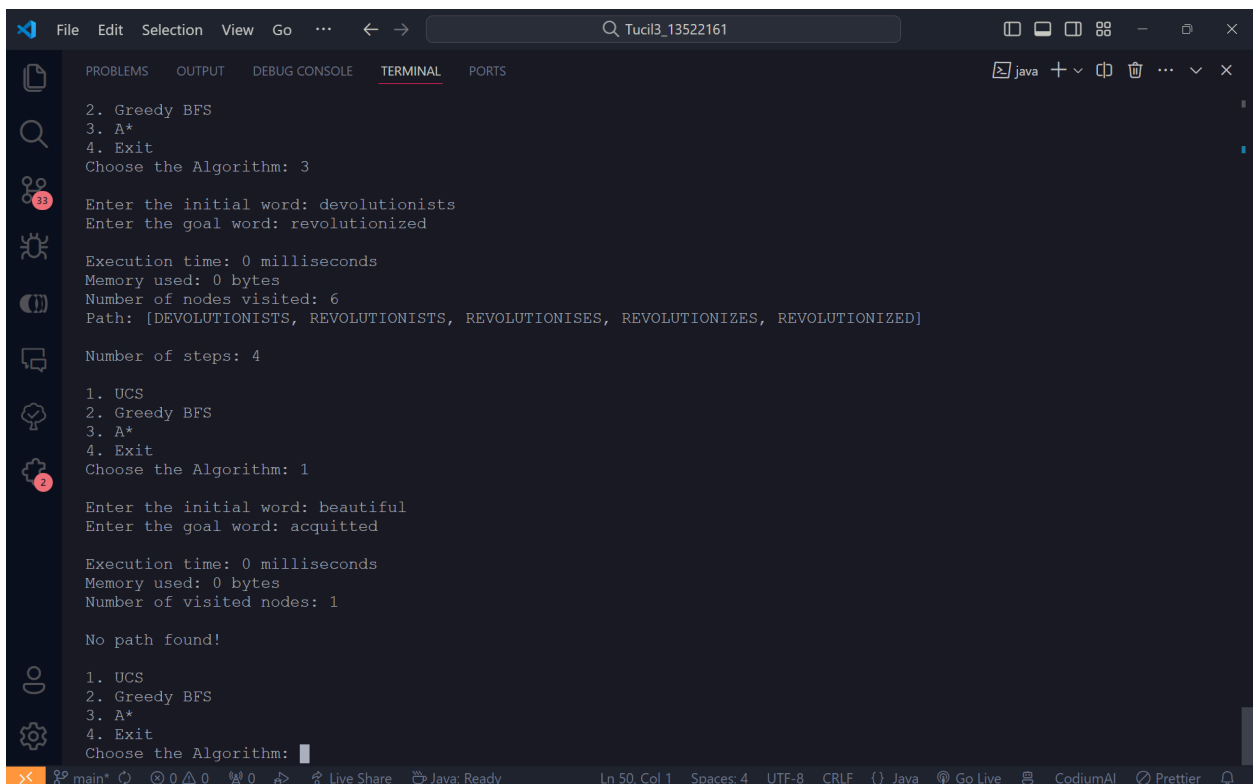
Enter the initial word: devolutionists
Enter the goal word: revolutionized

Execution time: 0 milliseconds
Memory used: 0 bytes
Number of nodes visited: 6
Path: [DEVOLUTIONISTS, REVOLUTIONISTS, REVOLUTIONISES, REVOLUTIONIZES, REVOLUTIONIZED]

Number of steps: 4

1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 
```

## 8. BEAUTIFUL -> ACQUITTED



```
File Edit Selection View Go ... ← → Q Tucil3_13522161
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 3

Enter the initial word: devolutionists
Enter the goal word: revolutionized

Execution time: 0 milliseconds
Memory used: 0 bytes
Number of nodes visited: 6
Path: [DEVOLUTIONISTS, REVOLUTIONISTS, REVOLUTIONISES, REVOLUTIONIZES, REVOLUTIONIZED]

Number of steps: 4

1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 1

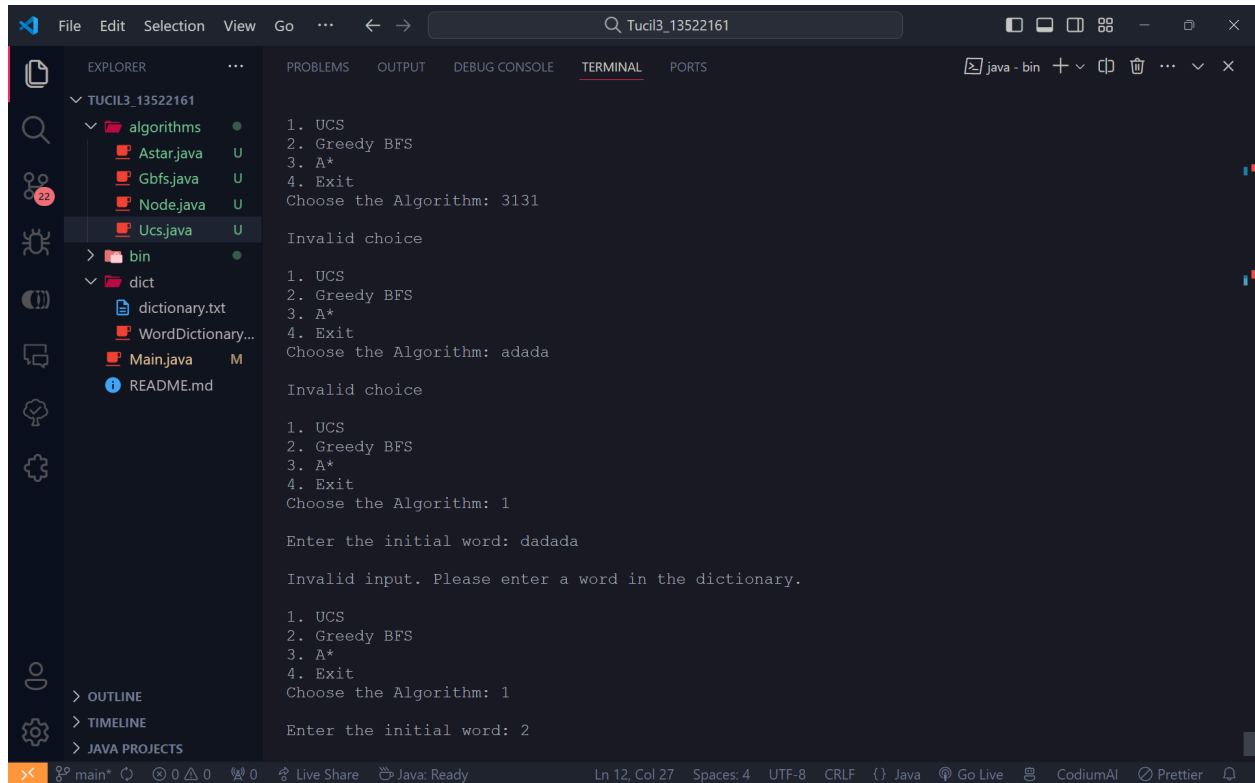
Enter the initial word: beautiful
Enter the goal word: acquitted

Execution time: 0 milliseconds
Memory used: 0 bytes
Number of visited nodes: 1

No path found!

1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 
```

## 9. Edge cases



The screenshot shows the VS Code interface with the Explorer view on the left and the Terminal view on the right. The Explorer view shows a project named 'TUCIL3\_13522161' with a folder 'algorithms' containing files 'Astar.java', 'Gbf.java', 'Node.java', and 'Ucs.java'. The Terminal view shows the output of the program, which is a Java application. The output displays a menu with four options: 1. UCS, 2. Greedy BFS, 3. A\*, and 4. Exit. The user enters '3131' and the program outputs 'Invalid choice'. The user enters 'adada' and the program outputs 'Invalid choice'. The user enters '1' and the program outputs 'Enter the initial word: dadada'. The user enters 'dadada' and the program outputs 'Invalid input. Please enter a word in the dictionary.'. The user enters '1' and the program outputs 'Enter the initial word: 2'.

```
1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 3131

Invalid choice

1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: adada

Invalid choice

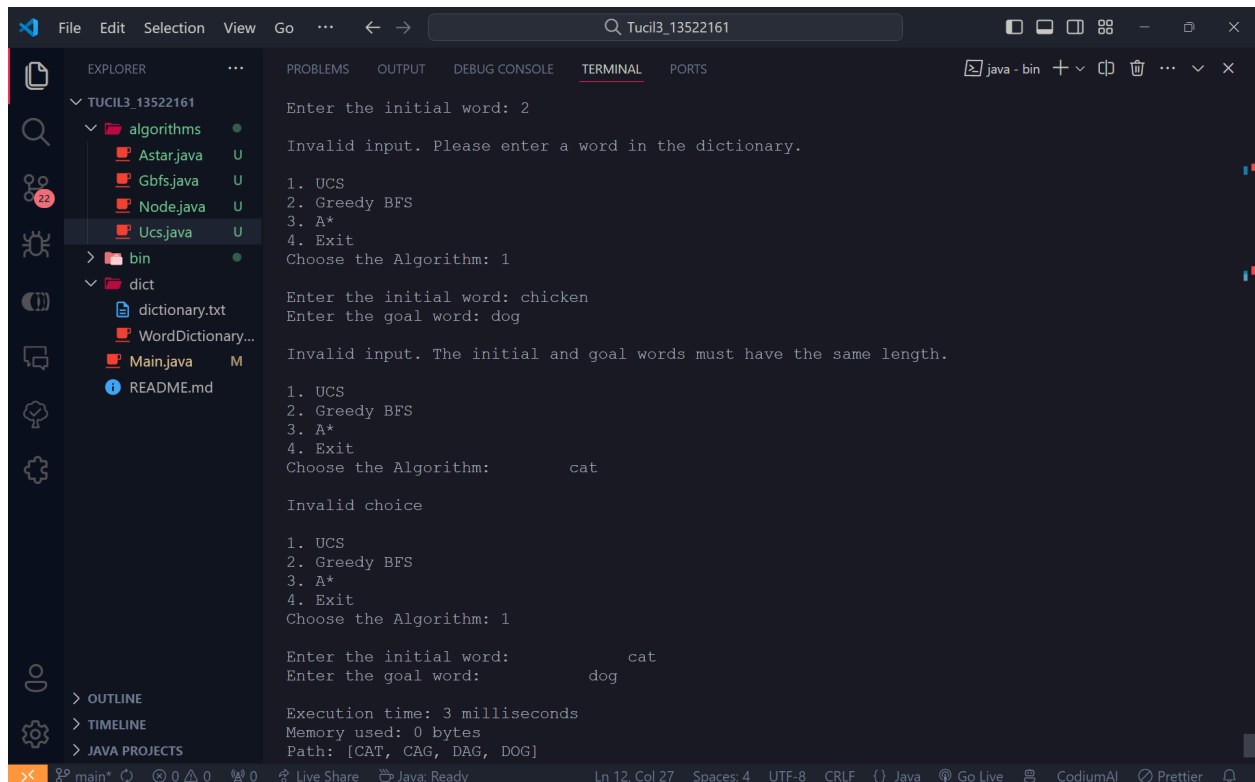
1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 1

Enter the initial word: dadada

Invalid input. Please enter a word in the dictionary.

1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 1

Enter the initial word: 2
```



The screenshot shows the VS Code interface with the Explorer view on the left and the Terminal view on the right. The Explorer view shows a project named 'TUCIL3\_13522161' with a folder 'algorithms' containing files 'Astar.java', 'Gbf.java', 'Node.java', and 'Ucs.java'. The Terminal view shows the output of the program, which is a Java application. The output displays a menu with four options: 1. UCS, 2. Greedy BFS, 3. A\*, and 4. Exit. The user enters '1' and the program outputs 'Enter the initial word: 2'. The user enters '2' and the program outputs 'Invalid input. Please enter a word in the dictionary.'. The user enters '1' and the program outputs 'Enter the initial word: chicken'. The user enters 'dog' and the program outputs 'Invalid input. The initial and goal words must have the same length.'. The user enters '1' and the program outputs 'Enter the initial word: cat'. The user enters 'cat' and the program outputs 'Invalid choice'. The user enters '1' and the program outputs 'Enter the initial word: cat'. The user enters 'dog' and the program outputs 'Execution time: 3 milliseconds'. The program also outputs 'Memory used: 0 bytes' and 'Path: [CAT, CAG, DAG, DOG]'.

```
Enter the initial word: 2

Invalid input. Please enter a word in the dictionary.

1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 1

Enter the initial word: chicken
Enter the goal word: dog

Invalid input. The initial and goal words must have the same length.

1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: cat

Invalid choice

1. UCS
2. Greedy BFS
3. A*
4. Exit
Choose the Algorithm: 1

Enter the initial word: cat
Enter the goal word: dog

Execution time: 3 milliseconds
Memory used: 0 bytes
Path: [CAT, CAG, DAG, DOG]
```

## BAB 4

### ANALISIS

Dari Test Case di atas, bisa dilihat perbedaan performa masing-masing algoritma dari segi waktu, memory, dan path yang dihasilkan. Pada algoritma UCS, selalu menghasilkan path yang optimal (hasil dibandingkan dengan [Word Ladder Solver \(ceptimus.co.uk\)](http://ceptimus.co.uk)). Namun, waktu dan memory yang dipakai selalu lebih besar dibandingkan dengan A\* walau path yang dihasilkan sama, ini karena UCS akan mengunjungi lebih banyak nodes karena hanya mempertimbangkan jarak dari node ke root tanpa mempertimbangkan jarak dari node ke goal, dengan mempertimbangkan jarak dari node ke goal ( $h(n)$ ), seperti pada A\*, jumlah nodes akan diseleksi kembali sehingga tidak akan mengunjungi node yang tidak berpotensi memberikan hasil yang optimal.

Hasil dari GBFS lebih cepat dan membutuhkan lebih sedikit memory dibandingkan dengan UCS maupun A\*, hal ini bisa terjadi karena GBFS menggunakan heuristik untuk memilih node berikutnya yang akan dikunjungi berdasarkan perkiraan jarak ke tujuan. Ini berarti GBFS cenderung memilih node yang tampaknya paling dekat dengan tujuan, tanpa mempertimbangkan biaya total jalur yang telah dilalui. Dengan pendekatan ini, GBFS bisa mencapai tujuan lebih cepat dalam beberapa kasus karena secara langsung "menuju" tujuan. Selain itu, karena GBFS tidak mempertimbangkan biaya total jalur, ia tidak perlu menyimpan dan memperbarui informasi tentang biaya jalur untuk setiap node. Ini berarti GBFS bisa menggunakan lebih sedikit memori dibandingkan UCS dan A\*, yang harus menyimpan dan memperbarui informasi ini. Namun, GBFS juga dapat stuck pada local minimum atau plateau, yaitu heuristic value berjumlah sama untuk semua nodes yang dicari, membuat algoritma ini tidak tahu ingin pilih path yang mana. Hal-hal tersebut membuat time and space complexity dari GBFS lebih rendah dengan mengorbankan path yang tidak selalu optimal, karena prinsip dari GBFS ini berharap solusi optimal pada lokal dapat menghasilkan solusi optimal pada global, yang tidak selalu terjadi. Bisa dilihat pada test case di atas, semakin banyak nodes yang dikunjungi, akan semakin rentan GBFS tidak menemukan solusi optimal.

GBFS berharap pada maksimum lokal bisa dilihat pada test case yang kedua, pada UCS dan A\*, path yang dituju adalah dari MAKE menuju MALE atau MASE pada A\*, namun pada GBFS adalah MAKE menuju FAKE, karena FAKE ke tujuan FLAT memiliki satu huruf sama,

namun MALE dan MASE yang dipilih oleh UCS dan A\* memiliki 0 kesamaan dengan tujuan FLAT, maka dari itu GBFS memilih FAKE, walaupun path yang dihasilkan lebih panjang dari optimal, berarti harapan pada lokal tidak sejalan dengan optimal pada global.

Algoritma A\* menjadi algoritma yang paling bagus dari segi waktu, space, dan juga path yang dihasilkan pada test case di atas, hal ini terjadi karena A\* menggabungkan kedua algoritma UCS dan GBFS menjadi satu heuristic value, yang mana heuristic akan semakin bagus dalam memilih node yang ingin dikunjungi. Berikut perbandingan time dan space complexity dari ketiga algoritma ini:

1. A\*:

- Kompleksitas Waktu: Pada kasus terburuk, A\* harus mengeksplorasi semua node dalam ruang pencarian, sehingga kompleksitas waktunya adalah  $O(b^d)$ , di mana  $b$  adalah faktor percabangan (jumlah rata-rata node anak) dan  $d$  adalah kedalaman solusi.
- Kompleksitas Ruang: A\* harus menyimpan semua node yang dihasilkan tetapi belum dieksplorasi dalam memori, sehingga kompleksitas ruangnya juga  $O(b^d)$ .

2. UCS:

- Kompleksitas Waktu: Sama seperti A\*, UCS juga harus mengeksplorasi semua node dalam kasus terburuk, sehingga kompleksitas waktunya adalah  $O(b^d)$ .
- Kompleksitas Ruang: UCS juga harus menyimpan semua node yang dihasilkan tetapi belum dieksplorasi dalam memori, sehingga kompleksitas ruangnya juga  $O(b^d)$ .

3. GBFS:

- Kompleksitas Waktu: GBFS bisa sangat cepat jika heuristiknya baik, tetapi dalam kasus terburuk (misalnya, jika heuristiknya buruk atau tidak ada solusi), GBFS juga harus mengeksplorasi semua node, sehingga kompleksitas waktunya adalah  $O(b^d)$ .
- Kompleksitas Ruang: GBFS hanya perlu menyimpan node yang dihasilkan tetapi belum dieksplorasi dalam memori, sehingga kompleksitas ruangnya adalah  $O(b^d)$ .

Perbandingan kasar ini, terlihat sama semua untuk complexity nya, namun pada praktik, hasil dapat berbeda jauh.

Bisa dilihat juga pada edge cases di atas, program sudah dibuat agar tidak dapat terminate kecuali user menginginkannya, seperti handle input mismatch, kata tidak ada dalam dictionary, hingga ketika tidak terdapat path, dan segala exception sudah dihandle dengan try catch ataupun metode lainnya pada program.









Program ini juga menggunakan dictionary yang digunakan oleh pranala [Word Ladder Solver \(ceptimus.co.uk\)](#), yang mana digunakan juga di kejuaraan scrabble dunia, meminimalisasi adanya ketidakcocokan dengan dictionary lain yang memengaruhi hasil pencarian.

## **BAB 5**

### **PRANALA REPOSITORY**

Github: [https://github.com/akmalrmn/Tucil3\\_13522161](https://github.com/akmalrmn/Tucil3_13522161)

## LAMPIRAN

Poin	Ya	Tidak
1. Program berhasil dijalankan.		
2. Program dapat menemukan rangkaian kata dari <i>start word</i> ke <i>end word</i> sesuai aturan permainan dengan algoritma UCS		
3. Solusi yang diberikan pada algoritma UCS optimal		
4. Program dapat menemukan rangkaian kata dari <i>start word</i> ke <i>end word</i> sesuai aturan permainan dengan algoritma <i>Greedy Best First Search</i>		
5. Program dapat menemukan rangkaian kata dari <i>start word</i> ke <i>end word</i> sesuai aturan permainan dengan algoritma A*		
6. Solusi yang diberikan pada algoritma A* optimal		
7. <b>[Bonus]:</b> Program memiliki tampilan GUI		